VULNERABILITY WRITEUP:

One of the vulnerabilities is that Snapitterbook is vulnerable to a stored XSS attack. The idea behind a stored XSS attack is that an attacker stores a malicious script in the web server/website, so that when the victim opens the browser in the same place as the malicious script, the browser will unknowingly run the script in the same origin as the server therefore leading to negative consequences such as leaking information or having the victim performing unwanted actions. I was able to execute a stored XSS attack in the "post" page of Snapitterbook. Looking at the server code, we can see that in the "post" function, the server makes sure that a valid user who is logged in is posting to a wall. After verifying the user, the function collects the input the user wants to post but only escapes the input with the method escpae_sql which only escapes certain characters such as semi-colons, dashes, and single quotes. The vulnerability here is that the post function does not make use of the escape_html method which escapes inequality symbols. Because of this, I was able to input the following script onto my user's wall (xoxogg's wall): <script> alert(document.cookie) </script>. What this script does is that now anytime any user visits xoxogg's wall, their browser will execute the JavaScript code placed on xoxogg's wall, and in my specific case the script will now steal the cookie of that user, thus allowing xoxogg to now log in as that user. This can happen because this stored XSS attack subverts the same origin policy. Because the post function in the server code does not fully escape all malicious characters, an attacker can exploit this by injecting any malicious javascript that does not make use of the characters escaped in escpe_sql. So various XSS attacks could be performed there. An easy fix could be to instead of just making use of the escape_sql method, also make use of the escape_html method so then no javascript could be posted on walls. An even better fix would be to do a whitelist sanitization so that input can only be allowed to contain certain specific characters such as maybe letters and numbers and some punctuation marks for example. With whitelisting we are helping prevent any malicious characters from being posted that could cause any future XSS attack.

Another vulnerability is that Snapitterbook is vulnerable to a SQL Injection attack. The idea behind a SQL Injection attack is that an attacker can execute a malicious SQL query that can control the website/server's database in order to leak and gain valuable information. I was able to execute a SQL injection attack in the "profile" page of Snapitterbook. If we take a closer look at the server code, we can see that there are two SQL commands, one that updates a user's avatar in the database and the other updates a user's age in the database. The vulnerability here can be found in the SQL command for age. If we look at any other SQL command in the server code, we notice that all input will be surrounded by single quotes then placed into the database. For example:
"UPDATE users SET age={} WHERE username='{}';".format(age, username) . In this SQL command, we can see that a user's username will be surrounded by quotes THEN will be placed into the database. But, the vulnerability here is that the value that is inputted into age is not surrounded by quotes because the SQL command is expecting an integer. Instead of inputting an int for age, an attacker can place a malicious SQL command that can extract anything from the database or even remove things from the database. An attacker can place a SQL command that collects the session ID for all users using Snapitterbook or could even drop the entire table in the database. Because a SQL command can sometimes be written using only letter characters, an attacker who inputs a malicious SQL command in "age" can avoid all of the escaping from both the escape_sql and escape_html methods in the server code and will be directly executed while the server is updating the user's age. The only thing an attacker must

do beforehand on the website is change the type of the “age” label in the HTML code to accept text rather than solely accept numbers as input in order to successfully inject a SQL command in “age” on Snapitterbook. One fix for the SQL injection vulnerability is to change the SQL command in the server code so that a user’s age will be surrounded with quotes just like the rest of the SQL commands in the server code when updating it in the database through the profile function. Another fix is also whitelisting the input since “age” only needs numbers. This way, we can constrain the type for “age” to only consist of integers which will reject any SQL commands or anything malicious other than numbers.

WEAPONIZE VULNERABILITY:

One exploit that will have minimal user interaction from dirks is performing a SQL Injection attack. I already explained how to perform a SQL injection in the vulnerability write up but to summarize the SQL Injection attack, the vulnerability is that when a user wants to update their age, the server code updates their age by executing a SQL query but does not surround the input for age in single quotes, so if an attacker were to place a malicious SQL command into age, it will be executed. So to go further into detail about the weaponized vulnerability, first we will need to change the HTML code on the “profile” page so that age can accept more than just numbers, so we change the age label type to be text so that any character could be inputted into age. Next, we will need to come up with a SQL command that gets the session ID of the user dirks. After looking at the server and database code, we can see that both the session ID and username are stored inside the table in the database called “sessions” upon logging in. So as long as dirks is logged in, this SQL Injection should work. An attacker can use the following SQL command: **SELECT (id uuid, username) FROM sessions** and upon execution, the SQL command will have allowed the attacker to view on his wall (where age should be) the username and session ID of everybody who is logged in. After taking note of dirks’ session ID, an attacker can now log out of their account, replace his own cookie on Snapitterbook to hold the session ID of Dirks, and once the cookies are set, the attacker can click on “login” and Snapitterbook will say that the attacker is already logged in, thus confirming that the attacker is now fully in control of dirks’ profile. Once in control of dirk’s profile, the attacker can now create a post which appears to be from dirks, change his avatar, and change his age, anything dirks could do the attacker can now do.

OTHER ISSUES:

One of the issues involves the function escape_sql. The problem with escape_sql is that an attacker can still craft a string that will be valid Javascript/SQL after escaping certain characters. For example if an attacker would be able to somehow inject a “--” (double dash) into a script, it would cause the rest of the line/input to be ignored thus causing some problems when inputting data into the database. For example, when the server code involves the execution of a SQL command that involves two or more variables (such as in the profile function it needs an age and username: "UPDATE users SET age={} WHERE username='{}';".format(age, username)) if an attacker were to somehow place a “--” after inputting a number for age, then nothing will be stored for username because the “--” causes the rest of the input to be ignored. Since all “--” are “escaped” in escape_sql, an attacker can just replace “--” in their input with “-;-” because in escape_sql, the semicolon is replaced with a blank so “-;-” ends up becoming “--” after escape_sql. So any attacker can execute an attack such as putting in a random value into age followed by a “-;-” to cause an error in storing data in the database. So basically, the vulnerability here is that the

“escaping” that is being performed in escape_sql in the server code still allows attackers to cleverly craft malicious input. A fix to this is that instead of replacing unwanted characters with other characters (for example replacing “;” with “”) in escape_sql or escape_html, the server code should actually escape malicious characters by using backslashes in front of special characters. By doing this, we can prevent malicious input from being executed. Another fix would be to whitelist input by only allowing certain characters to be used such as maybe only allowing letter characters, numbers, and some punctuation marks so we can prevent any unwanted characters that might cause harm.

Another issue is found in the HTML of the website. What I mean by this is that anybody who is on the website can easily inspect the website and edit the HTML there and change any values they want to make in order to make the website look how they want. An attacker can take full advantage of this and cause some serious damage by changing usernames, or ages, or even avatars to their liking so that the database can change information. In order to fix this vulnerability on this website, one of the developers of Snapitterbook can temporarily remove the HTML DOM when inspect is opened. So when the website detects when the inspect HTML editor/debugger is opened, the website removes the DOM/code and stores the code in a variable and when the debugger is closed, the website returns all of the code/DOM from the variable. By doing this, the website is guaranteeing minimum user edits on the website that could cause any false user input being inputted to the website or harmful changes to the website.

Another issue with Snapitterbook is that all of the functions that include SQL commands such as the functions login, profile, get_user_info, and etc are all based on raw user input. If not sanitized correctly, user input from attackers going into the SQL commands in the server code can harm the system into leaking information or causing other users to to perform harmful actions unwillingly. In order to fix this issue, Snapitterbook should avoid building a SQL command in all of its functions in the server code based on raw user input, and should instead use existing tools or frameworks that have built in sanitization and protection for other common vulnerabilities. An example that was discussed in lecture is the Django web framework which defines a query abstraction layer which sits over SQL and allows applications such as Snapitterbook to avoid writing raw SQL by taking a SQL query and replacing user inputs with escaped values. This will prevent attackers from learning any information from the database through input into SQL commands.